COMPUTER PROGRAM FAULT DETECTION AND CORRECTION

Sheldon Lee Margolis

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

COMPUTER PROGRAM FAULT DETECTION

AND CORRECTION

by

Sheldon Lee Margolis

Thesis Advisor: V. Powers

June 1972

T14155

Approved for public release; distribution unlimited.

Computer Program Fault Detection and Correction

by

Sheldon Lee Margolis
Lieutenant, United States Navy
B.E.E., Rensselaer Polytechnic Institute, 1965

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 1972

Thesis
M3425
c.1

ABSTRACT

A major problem with present fleet tactical computer programs is the failures that occur because of software faults caused by intermittent transient errors. Presented are measures that should be taken to prevent the generation of these transients, and possible steps for the detection and the correction of faults caused by these transients. Examples are drawn from the practices used in the SAFEGUARD, AEGIS, and STAR computer system programs. The paper describes some of the causes of program degradation and requirements for future activities to overcome these problems. Some specific examples are shown to illustrate corrective measures that could be taken. The applicability of protective redundancy to computer programs is established.

TABLE OF CONTENTS

I. INTRODUCTION 4

II. FAULT TOLERANT HARDWARE 8

III. MONITORING AND MEASUREMENT OF SOFTWARE 21

IV. FAULT-FREE RELIABLE PROGRAMS 26

V. FAULT DETECTION AND CORRECTION IN COMPUTER PROGRAMS 36

VI. CONCLUSIONS 48

LIST OF REFERENCES 52

INITIAL DISTRIBUTION LIST 54

FORM DD 1473 55

I. INTRODUCTION

The United States Navy has now embarked on a program that will soon see all major tactical and support systems driven, controlled, and maintained by digital computers or by information provided by digital computers. To insure that these systems will be used as they are designed, senior personnel in the key decision making positions must be given assurance that the system will run properly, and that the system will indicate to the operator when it is not performing properly so that corrective steps can be taken to restore the system to its correct operating level.

There are two classes of computer system faults that can cause degradation, and either will lead to failure if the minor faults are not periodically or continuously corrected. They are the hardware components and the software programs of the computer system. The first of these areas (hardware) is currently attracting the most attention, and is the one in which the most progress is being made. The software segment is the least talked about, but to the Navy it is equally important. While in many computer system operations the failure of the software can cause a loss of time, or slowdown in production, the loss of a tactical program when it is needed will result in the loss of the ship and whatever it is defending. The problems of computer software failure are widespread and apply

equally to the batch processing of user programs and to the real-time and time shared environment of tactical modules. In the chapters to follow several possible causes of program failures will be described. The effects of protection systems now in use will be identified and possible additional approaches will be outlined. Evidence of the problem in today's tactical environment is based on information received from officers and agencies associated with the Navy Tactical Data System (NTDS), as well as trouble reports generated about this system. The reports indicate that most of today's failures are occurring in the computer software segment of the system. Since the problem is usually thought to be caused by a specific function either using an un-debugged path or becoming erratic in its operation. The cure used is to reload the computer programs and restart, without gathering data on why the failure actually occurred.

The research process involved talks with Raytheon, Computer Science Corporation (CSC), and RCA programmers writing the operational program modules for the AEGIS Weapon System; with UNIVAC engineers and programmers who are working on the AN-UYK/7 computer and CMS-2 compiler; with field engineers from IBM; and visits to the NTDS Test Site at Mare Island and the Fleet Computer Programming Center, Pacific in San Diego. In addition Bell Laboratories in Madison, New Jersey provided information on the steps that they are taking to make the ABM system programs fault tolerant.

The method of attack that is used in the development of software reliability is similar to that used by hardware designers in that the first step taken is to insure that the initial program is designed correctly, put together correctly, and will run fault free if no external stimuli are introduced to the module.

Chapter II describes the present state-of-the-art techniques being used in the design of fault tolerant computer hardware. An example (JPL STAR Computer) of the techniques is given to provide a background of redundancy methods that can be applied to computer software. In Chapter III, monitoring and measurement of computer software performance is described. Information for detecting and correcting faults in programs will be gathered by using suitable combinations of these techniques. Methods of organizing and checking out computer programs to insure that they are fault free are described in Chapter IV. A summary of program proving by techniques other than executing with test cases is given and the type of checkout being used in state-of-the-art development is illustrated (SAFEGUARD Computer Programs). Chapter V describes the methods of fault detection that are available, presents new areas for consideration and delineates their applicability to Navy computer systems. The need for the use of redundancy techniques for computer programs is shown. A summary of what has been accomplished and directions for future work is given in Chapter VI.

When the research for the thesis showed that the ground to be covered was larger than anticipated, a specific endpoint for this development was selected. The paper presents the work I have done in establishing some of the causes for program degradation, and establishing direction for future activity to overcome these problems. Some specific examples are shown to illustrate some corrective measures that could be taken. The applicability of protective redundancy to computer programs is established.

II. FAULT TOLERANT HARDWARE

The inclusion of fault tolerance in the design of a computer would be unnecessary if it were possible to build a computer that was in complete agreement with the designer's intentions and would never wear out. Since this is not possible, the best that can be done is to consider fault tolerance in the initial stages of the design, when it is cheapest and easiest to implement, and allow it to be an inherent part of the computer.

There are three major sources of trouble that must be considered when designing and building a fault free or fault tolerant computer. These are initial design and wiring errors, hardware faults, and transient faults [Ref. 9].

The design and wiring errors exist in the hardwired program in the computer and in its mechanical connections. The validation and detection problem for this trouble is relatively simple as the errors generated are wide spread and easy to spot. In the hardwired programs, the algorithms implemented are usually of limited complexity and easy to follow. The mechanical connection errors are detected by logic diagnosis and testing techniques that have become widespread in circuit construction testing where backplane wiring and integrated circuitry are used. The hardest errors to detect in this class are

those where an incorrect result is obtained only for an infrequently occurring set of data, or if the fault is localized.

As defined by Avizienis [Ref. 6], hardware faults come in two categories. Hard failures are those which are permanent. In this case the element involved fails and continues to give either a "stuck at 0" or a "stuck at 1" output. It is also possible for this type of "stuck" failure to give both "1" and "0" outputs, but always at the wrong time. This failure usually occurs during burn-in[1] or after component wearout[2] has started. An intermittent fault is one that occurs for a certain set of inputs only, and causes the same type of "stuck" outputs as the hard failures. In both hard and intermittent component failures, the immediate symptom is a logic fault in the operation of the computer. That is, the current instruction will not be executed correctly or an incorrect result will be computed.

The transient fault is the third of the basic problem classes. This fault will be caused by the temporary incorrect operation of a component, or, more commonly, because of external interference with the computer. Such interference can be caused by irregularities in power supplies, electro-magnetic radiation hazards, environmental

---

[1]Burn-in. The operation of items prior to their ultimate application intended to stabilize their characteristics and identify early failures.

[2]Wearout. A failure that occurs near the end of the operability curve due to the deterioration process.

shocks, and similar events. This is the type of fault that tends to go undetected and uncorrected. There is usually no record of the failure having occurred and periodic checkout of the computer will not indicate that anything is wrong.

All of the types of failures described have one of two effects on the computer system. The fault may be independent -- affecting only the component or logic circuit it is in, or the fault may be catastrophic -- causing the entire computer system to fail. Much work is being done in computer design to combat these failures and to achieve a high degree of fault tolerance. The computer designer is increasing his use of the theory and practices of protective redundancy. This thesis begins to apply to computer program design and automatic repair the theory of protective redundancy that is being applied to hardware.

Protective redundancy consists of all additional programs, additional circuits and repetitions of operations that would not be needed in a perfect computer. The elements of protective redundancy must be applied in order to effect recovery. Throughout, "complete recovery" will be used to mean the continuation of system operation without loss of data [Ref. 8]. There are different degrees of recovery dependent upon the amount of protective redundancy that is used. The degrees are: 1) complete-the level of operation is unchanged; 2) restart-the level of operation is the same but the system picks up at a different point of operation, the program is "rolledback" to the last successful operation; 3) degraded-the computer continues running,

but with limited capabilities and perhaps at a different program location; 4) terminated-the computer has saved as much information as possible, has notified the operator of the failure, but is unable to continue program execution. The degree of recovery possible is dependent upon the severity of the fault and upon the amount of redundancy used. The amount of redundancy is limited by the amount of money, size, weight, and time that is available during development, and by the time that is available during computer operation for running redundancy programs. The redundancy in the system must be designed in to be purposeful and useful rather than accidental.

In hardware terms there are two types of redundancy that are most common: static-where the redundant units are always powered and connected; dynamic-where the standby unit is switched into the system and the failed unit switched out when the fault is detected. The two methods will be described in the following paragraphs, a comparison of the reliabilities obtainable will be shown, and the state-of-the-art will be exemplified by a description of the JPL STAR Computer (Jet Propulsion Laboratory Self-Testing and Repairing).

Concurrent with the design of computers, redundancy techniques have also been developed [Ref. 10]. Static or masking redundancy is the terminology applied to those methods that involve encoding of the function, active performance of all components of the system, and implicit recognition of the error. Using these techniques, the presence of a faulty element is immediately covered up by its

replacement element which has been permanently connected and concurrently running in the system. The static technique allows for replication from the component level to the system level. Triple modular redundancy (TMR) as defined by J. von Neumann [Ref. 18] provides for the availability of voting[1] at selected interfaces, with the faulty elements being dropped from the system until repaired. The advantages of the static approach are:

1) The corrective action is hardwired into the system and is thus immediate. This is particularly important if there is a high ratio of hard failures to transient errors.
2) During system operation there is no need to devote computer time to the running of fault diagnosis programs.
3) If the computer was not initially designed with useful redundancy, the conversion to static redundancy is straightforward and simple.

Dynamic or standby redundancy makes use of functionally identical units. In this organization, only the modules actually in use are "powered up". The recognition of failures in the dynamic method is usually explicit. Since the switching in of the spare unit must be automatic if the computer is to be self-repairing, two principal

---

[1] Voting. The comparison of the output of three or more units with majority rule and minority failure indicated.

methods of automatic response are used: 1) replacement of the faulty element by using one of the standby spares; 2) reconfiguration of the system into one that can still operate, but in a degraded mode. In both techniques, a diagnostic program must be present, and in addition the second method also requires the presence of a reconfiguration program. Error correction is accomplished by recomputation from the failed point or by program rollback[1].

The advantages of the dynamic redundancy techniques are [Ref. 6]:

1) Power is required for only one copy of each replaceable module required for the system, and the unpowered copies have inherently lower failure rates than the powered copies.
2) Fault isolation between subsystems is provided by the replacement switch, which prevents the spread of possible catastrophic failures.
3) All spares can be utilized where required, unlike the static case which can use each spare only where it is wired into the system.
4) The design of the individual replaceable components can be modified or improved, and the number of

---

[1]Program rollback. Returning by monitor system control, to the last point in the executive program that was executed correctly and continuing operation from that point.

spares increased or decreased to meet the needs of a given mission without changing the system design.

5) There is no need to increase the fan-in/fan-out requirements of the components as is often the case for logic elements used in static redundancy.

6) Pre-mission checkout is easier as not all of the replicated units are connected to the system and can be individually tested by use of diagnostic programs as opposed to the static case.

7) The need for synchronization required by static TMR techniques is not required using the dynamic system.

8) Dynamic redundancy has the ability to correct errors caused by transient faults by the use of program rollback.

The present "best approach" to the fault tolerant design problem is to use the advantages of both of these methods wherever possible. Since the dynamic techniques give a greater improvement in reliability than the static techniques [Ref. 2], more of the dynamic methods are being used in present systems.

TMR is used with "back-up dynamic sparing"[1] in sections where faults must be corrected on an instantaneous basis. Even with this

---

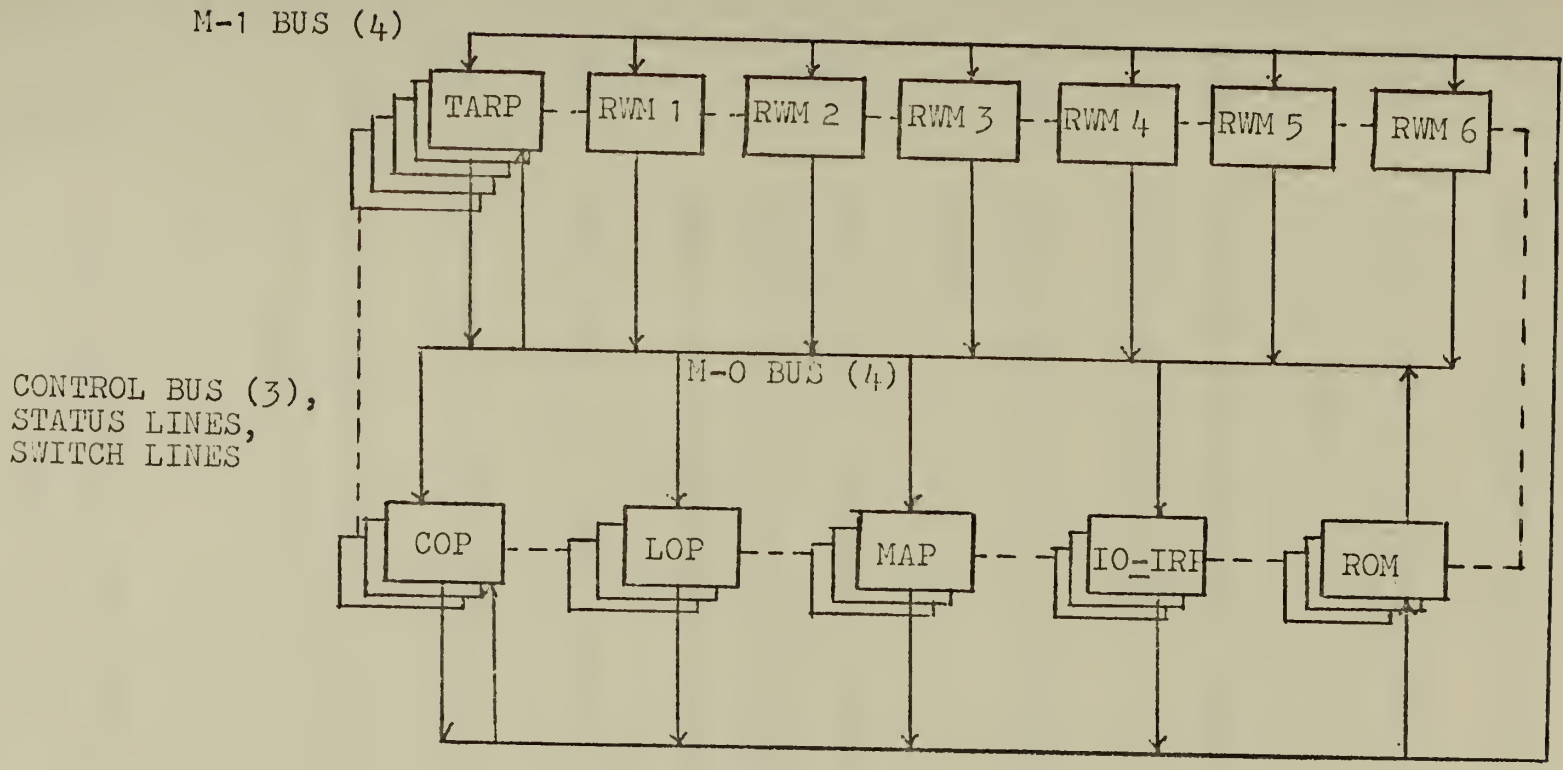
[1] The voting technique is used, with the units involved in the voting replaced using the dynamic technique of powering a standby unit when it is needed.

voting replacement, the possibility of a transient fault is allow for in the subsequent automatic checkout and return to spare status of the suspect unit.

JPL STAR COMPUTER

The most advanced of the fault tolerant computers is the JPL STAR Computer being developed at JPL at the California Institute of Technology [Ref. 2]. It makes use of a hybrid of the static and dynamic techniques. A block diagram (Fig. 1) of the STAR Computer shows the basic organization of the computer, while Table 1 describes the function of the modules shown in the block diagram. With the exception of the Test and Repair Processor (TARP), only one copy of each module is powered at a time. The TARP is the hybrid component, operating with basic TMR and standby sparing. The TARP controls synchronization in order to initiate recovery; otherwise the modules operate autonomously. The TARP is continuously informed as to how each of the powered units is operating. During normal operations the TARP is a monitor, and the stored programs are executed. The TARP has facilities to store "most recent" correctly executed processes so that it is able to perform rollback and restart after an error has been detected.

The TARP is the monitor in the STAR Computer and it controls the recovery mode. The TARP consists of a control and test segment (CAT) and a recovery control segment (REC). The CAT contains the



STAR COMPUTER ORGANIZATION

FIGURE 1 [REF 2]

COP Control processor, contains the location counter and index registers and performs modifications of instruction addresses before execution.

LOP Logic processor, performs logical operations on data words (two copies are powered).

MAP Main arithmetic processor, performs arithmetic operations on data words.

ROM READ-ONLY memory, 16,384 permanently stored words.

RWM READ-WRITE memory unit with 4096 words of storage (at least two units powered; 12 units are directly addressable, maximum 49,152 words).

IOP Input/output processor, contains I/O buffer registers.

IRP Interrupt processor, handles interrupt requests.

TARP Test and repair processor, monitors the operation of the computer and implements recovery (three copies powered).

STAR COMPUTER MODULE DESCRIPTION

TABLE 1 [Ref. 2]

standard mode control logic and prediction logic which predicts what the status symbols from the operational units should be. It performs fault detection, fault location, stops the computer and transfers control to the REC segment. The REC contains the "rollback point" address register for the resumption of operation after recovery. Upon receipt of an error message from the CAT segment, the REC segment orders the functional units to reset and attempts execution from the "rollback point". A repeated error results in the REC segment ordering the failed unit to be replaced by one of the standby spares.

Table 2 compares the reliability predicted for the STAR computer compared with that obtained by the Mariner Mars 1969 computer (MM 69) and a simplex computer of equivalent performance [Ref. 2]. The "K" in the table is the ratio of failures in powered units to failures in unpowered units. $K = \infty$ means no failures in unpowered units, $K = 1$ means there is an equal failure rate between powered and unpowered units. Table 3 shows the comparison of the amount of time each computer would be able to perform and still meet a specific mission reliability. As can be readily seen, the STAR is an order of magnitude or more better than the simplex computer, and 3-4 times better than MM 69 that made use of static redundancy techniques. Work is still being continued with the STAR Computer, with each improvement indicating new classes of concern (programs, switching speed, etc.).

Mission Time (h)	MM'69 Computer	Simplex Computer	STAR Computer with S Spares Upper Bound (K = ∞) S = 3	S = 2	Lower Bound (K = 1) S = 3	S = 2
4368 (6 months)	0.928	0.82	0.9999998	0.99997	0.999995	0.99982
43 680 (5 years)	0.475	0.14	0.997	0.97	0.966	0.87
87 360 (10 years)	0.225	0.019	0.96	0.79	0.71	0.45

RELIABILITY VERSUS TIME

TABLE 2 [Ref. 2]

Desired Mission Reliability	Mission Duration in Years: MM'69 Computer	Simplex Computer	STAR Computer with S Spares Upper Bound S = 3	S = 2	Lower Bound S = 3	S = 2
0.9	0.7	0.3	12.5	7.5	6.7	4.5
0.8	1.5	0.6	16.0	9.7	8.5	6.0
0.7	2.4	0.9	18.5	11.7	10.0	7.0
0.6	3.5	1.3	20.5	13.5	11.3	8.3

MISSION DURATION VERSUS REQUIRED RELIABILITY

TABLE 3 [Ref. 2]

The biggest step that had to be taken came with the realization that the improvements that were being made in the hardware fault tolerance were placing requirements on the design of the computer programs to support these advances.

III. MONITORING AND MEASUREMENT OF SOFTWARE

The development of any fault tolerant system requires the designer to be able to measure and monitor the performance of the modules he has developed. All along the development path he is faced with the trade-offs between cost, size, speed and capabilities of the observing programs as opposed to the functional programs.

Monitors and measurers of software can take the form of hardware equipment or software programs, and each has its advantages. In either case, they should be designed into the system they are to be used with, rather than added on as an afterthought. Measurement and the use of the results obtained must be carefully evaluated to keep from reaching erroneous conclusions about the system being observed. As real-time systems increase in size and in complexity, the requirement grows for monitors that will enable the operator to find errors and allow the designer to correct the errors. In the past systems have been designed and only after failing to operate as anticipated, have they been subjected to monitoring and performance measurement.

To be a useful tool, measurement and evaluation must take place continuously throughout the lifetime of an operating system. The proper testpoints, both hardware and software, must be added to the system as time demonstrates their need [Ref. 3]. The system evaluator should be given the system designer, because in most cases, he is the only

one with sufficient knowledge of the system to properly place the measurement points and to evaluate the data that is gathered from the points.

There are three software techniques that are used for system measurement [Ref. 7]. They are simulation models, analytic models and internal system measurement. Simulation has the disadvantage that the data for the simulation must be obtained from some source. If it is obtained from the system to be simulated, then the designer, in many cases, will prefer to work with the actual system rather than the simulation. Since the simulation is not usually written by the system designer, he may again not want to use it, suspecting that it is not as current as the system is. In the case where the simulation is used to predict system performance before the system has become operational, data must be gathered from similar systems, and the confidence level of the evaluator again is not very high. As evaluators become more experienced in the use of simulations, and are able to decide what data is useable and what is not, the value of reports received from simulations is increasing. The important conclusion to be drawn about the simulation technique is that the simulation must be kept simple and must be developed early enough so that it can be kept current with its operational system.

The use of analytical techniques has not yet reached the state of development where their use can be widespread. Making use of simplifying assumptions, average values, etc., the models' primary

usefulness appear to be in the study of gross system effects. Analytical techniques have not yet begun to achieve the type of success reported by the simulation methods [Ref. 3]. Their use is still limited to the few people who understand the mathematics behind the method, and the proper simplifying assumptions that must be made.

Internal system measurement is the technique that is most widely used, though the evaluation of the data obtained is still very difficult. The three types of internal measurements are event counting, trace recording, and sample taking [Ref. 7]. These methods all try to answer the question of where did the time go that is not accounted for by proper system operation. The important point that must be remembered when evaluating the data is to use no preconceived ideas to place the lost time. Every system is different and few generalities can be drawn.

Event counting involves the incrementing of a counter each time a specific event is started. This method enables the experienced evaluator to determine if an event is occurring more often than it should, and thus reset some of the program parameters. There is no tie to either time or to sequences of events using this method, but it is the simplest of all techniques, taking very little processor time.

Event tracing is the next step in complexity from event counting. In this technique, each time an event is started, its name and the time that it starts, plus any additional desired information, such as user,

is also recorded. The trace provides information on job paths, queues, time, and time between events. Care must be taken to keep the methods uniform so that different traces can be compared to determine if system changes are improving or deteriorating the system. Much success has been obtained using these techniques.

The sampling technique involves entering the system at random intervals and recording all queue, register, and device status information that is desired. This technique is able to gather information both on the operating system and on user programs. Care must be taken to insure that the sampling is done in a random manner to avoid synchronization with the running programs that might tend to invalidate the data being gathered. Data reduction programs are generally used to analyze the data that is obtained.

It must be emphasized that the data gathered by any of the above techniques must be examined very carefully. It is easy to draw improper conclusions from looking at the data obtained and make changes to the operating system that will do more harm than good. In addition to these software techniques, hardware devices can be directly coupled to the computer system to record data that can be used in performance analysis.

Devices such as the IBM SPAR (systems performance activity recorder) can be coupled by probes to measure the logic pulses or levels at the points of interest of the system [Ref. 7]. Care must be taken when introducing additional hardware to keep from adding to the

system load. When properly used, the hardware measurement technique offers three major advantages over the previously mentioned software techniques:

1) Effectively creates no system interference.

2) Allows very fine measurements at the microsecond or smaller level, and obtains utilization percentages.

3) Allows access to all parts of the system.

All of the techniques mentioned turn out to be performance measurers and system monitors. These are the predominant methods that are being used in large computer complexes today. There are no clearly defined approaches to the monitoring of and the correction of system errors unless they are the error types that generate interrupts. The methods described must be extended before they become useful to a system operator in a realtime environment.

IV. FAULT-FREE RELIABLE PROGRAMS

When writing a large program, proper organizational and programming techniques must be used to obtain a program that is both fault-free and reliable. The first consideration is organization to avoid the generation of errors while producing a program [Ref. 12].

The development process is plagued with many problems that must be considered when organizing to produce a large system program. Unless proper documentation and validation techniques are followed, the resulting system will prove to be unmaintainable and useless. The primary areas of concern are [Ref. 12]:

1) Programmer turnover--throughout the development period of a large operating system, it can be anticipated that programmer turnover will average about 20% per year.
2) Hardware turnover--since the software is usually placed into development after the hardware is in existence, there is a requirement to modify the software as it is being written to account for changes that are being made in the hardware.
3) Software turnover--during the development of the system it is likely that new ideas and developments will require the system to be modified as it is being written.

4) Inexperienced programmers--aside from the actual designers, the group programmers will be unfamiliar with the system under development and the desired organization.

5) No single information source--no individual, including the system designer, is able to be totally familiar with the entire project.

6) Poor communication--between members of the same programming group, between groups, and between project managers.

7) Poor documentation--an individual will program a segment and understand what is accomplished, but anyone else will be unable to determine the operation of that particular module.

These problems must be successfully countered before the problems of program verification and reliability can be handled. The basic solution to the problems is to establish a computer program development plan that determines the approaches that are to be taken in organization, documentation, verification, and communication. The development plan will establish the order of the program modules, the method of review of the modules, and the level at which documentation is to take place. The document must establish the interfaces between modules and the time frame of the reviews and reports to the system managers.

It is the documentation effort that is of primary concern to the overall system coordinator. With proper documentation he is able to

get system flowcharts and listings that are understandable and that can be used in the debugging of the program. He is able to determine if the desired fault detection capabilities have been included in the module, and he is able to implement changes and improvements even in the face of turnover in personnel and equipment. With the solution of the introductory organizational problems, the system manager is then able to turn to the problem of program verification and validation. Here again there are two causes for concern, one relatively simple, and one that is still extremely difficult.

The first is the assurance that the program has been converted from thought to flowchart to code to an input medium without the introduction of mechanical errors. The second is keeping the copy of the program in the computer correct.

By checking and by human verification it is usually possible to get the program into the computer and begin the process of program verification and debugging. Another way of looking at this development process is to break down the procurement of a software system into steps, and see how far we have progressed. Some typical steps in this development are:

1) statement of the problem
2) description of a solution algorithm
3) composition and coding of the program into a suitable language

4) preparation and transfer of the code onto an input media

5) compilation of the program into binary machine code

6) execution using sample data

7) debugging using sample data and returning to step 3

8) final phase testing using typical data and looping to step 3 as required

9) confident issuing of the program to an operational environment

To this point the first four of these steps have been discussed, and in most systems they are the easiest to accomplish. It is the sources of errors in the remaining steps that must be detected and eliminated. Starting in the compilation phase, it is felt that more of the error detection should be automated for more completeness. A wider range of programming checking devices that could uncover improper coding and input format should be evoked by the compiler. This could include checks on variable declarations, range of subscripts for arrays, loop boundaries, etc., in an attempt to catch errors which most compilers miss and which are hard for the programmer to visibly see on a printout. In addition, making the compiler responsible for the catching of errors in program semantics, although complicating the compiler, would simplify the programmers' problems. The modules under development, after passing through this type of compiler, would thus substantially reduce the need to run with sample data and loop back as previously indicated.

The normal progression of events would have the software under development proceed to bulk testing with all types of expected sample data. In this way the system manager would attempt to prove to his own satisfaction that steps seven and eight were complete and that the program could be issued to its operational users. It is obvious from the results of system packages now being used that this method of verification is not successful. The programs that are in the fleet consistently have to be patched in order to fix bugs that were not uncovered during these test and verification periods. For this reason, work is being done in the development of automatic program checkout and verification.

Automatic Program Checkout

There are many approaches being taken in the field of automatic checkout, and several of these are briefly described below. For more complete descriptions, the references should be read.

1) Dijkstra and Naur [Ref. 16, 17] advocate a systematic approach to the writing of the program. Though not mechanical, the approach calls for working slowly from the top down, so that the program is proven to be correct at each step in its construction. Thus the written program would be correct and would meet all of its specifications. This attack would not be applicable to programs already written.

2) Manna and Waldinger [Ref. 15] use a mechanical theorem proving technique based on predicate calculus. The theorem represents the specifications of the program and the calculus is used to operate on it to prove the program correct. This mechanical approach is limited by the strength of the theorem prover and by what can be stated about the program in theorem format.

3) London [Ref. 13] attempts to take a program that is already written along with the specifications of what it is supposed to do, and certify by mathematical proof that the specification is met in all cases. During the proof, the specification may have to be altered or changes made to the program if the two do not match as required.

4) King [Ref. 4] attempts to prove that programs will execute correctly by establishing an abstract model for computations, then using predicate calculus to establish the correctness of the different paths through the program. The method may fail when not all verification conditions are true; however, it can be stated that if the program is correct, there exists some inductive predicates which will yield a proof by this technique.

In a paper presented by Elspas, et al, at the 1971 International Symposium on Fault-Tolerant Computing [Ref. 14], the relative merits of the different program verification methods are summarized.

The major difference that this paper draws between the methods is that for any large program, the hand calculations (1, 3, 4 above) become excessive in length and can generate their own errors that cancel out their effectiveness. On the basis of the comparisons and additional work, several conclusions are drawn and recommendations made:

1) There is going to be no rapid breakthrough in the establishment of verification techniques.
2) The use of off-the-shelf routines will improve the performance and ease of verification of programs.
3) Informal proofs will currently be more effective than using theorem provers until such time as the theorem provers become more suitable for computer use.
4) It is likely that any mechanical verification procedure will be isomorphic to a mechanical proof of the theorem corresponding to the program-assertion pair.
5) Present theorem provers are not convenient to use because of the low level of predicate calculus, making it difficult to describe even simple data structures. A new high level description and axioms are required.
6) Programs must be properly partitioned in order to prevent the overflowing of variables from module to module and thus making proofs extremely difficult.

7) Automatic program verification procedures are more likely to be more easily carried out - and are of more practical interest – for programs characterized by large numbers of simple operations.

Thus there is much to be done in the development of the techniques required for automatic program validation and verification. For systems that are currently under development, improvements and increased testing are being applied in order to develop programs that are as correct and as reliable as possible. As an example of the methods currently being , a brief description of the techniques being used by Bell Labs is presented. The procedure presented is being applied by the large development group that is working on the computer software for the SAFEGUARD project. The material was contained in a letter from T. Crowley, [Ref. 5], the Executive Director of the SAFEGUARD Design Division.

1) An extensive system of documentation including manuals, specifications, program documentation, etc., exists.
2) An advanced and flexible high-level programming language (CENTRAN) is used.
3) The techniques of Structured Programming, which involve GOTO-free, modular code are being applied. This causes the code to be more readable and thus simpler to maintain, as well as providing for more efficient testing.

4) Programmer Notes is a special document by which programmers can quickly and informally keep others on the project informed of new methods and ideas.

5) Program language reference cards, covering the major languages used on the project, provide for quick and easy reference to the information that is most needed by programmers.

6) An extensive training program for project members encompasses both hardware and software subjects.

7) Periodic design reviews throughout the software life cycle are used to make visible the requirements, specifications, design, and test and integration plans for constructive comment and critique.

8) Quality assurance checks on both documentation and listings are performed.

Programs are first coded, compiled, and then extensively unit-tested. The related programs are integrated with one another, and again completely tested by groups of programmers other than those who first worked on the modules. After the modules are integrated and checked out, a System Readiness Verification system which contains a System Exerciser, is used to thoroughly test the system as a whole. Testing is first done at a test site using prototype equipment, and is then duplicated on the actual equipment when delivered to the operational sites.

A special group is responsible for controlling all "frozen" programs. They maintain the official files and make any changes that are required. There is an extensive change management system to handle trouble reports and corrective reports which this group supervises. This is to ensure that all groups working on a module in the frozen segment have the same copy to work with. Important procedures for all steps are specified in the SAFEGUARD Policies, Procedures, and Standards Manual. This assures that all personnel on the project are working towards the same goals with the same tools.

Bell Labs is thus applying all available tools in order to provide a state-of-the-art computer program to run a state-of-the-art hardware system. Each of the areas outlined for ensuring program validation and reliability have been covered by Bell Labs in this production effort. The one class of errors that has not been covered in this presentation is errors that occur after the program is resident in the computer, and that is the subject of the next chapter.

V. FAULT DETECTION AND CORRECTION IN COMPUTER PROGRAMS

This area of computer faults and failures has received the least amount of attention until very recently as it is hard to detect this type of error in systems which have either low reliable hardware or improperly debugged software. It is only with recent developments that this type of failure begins to show up.

A major problem with fleet tactical programs is the failures that occur because of computer program faults. The situation is such that the most common cause of system failure is the computer program. While researching this problem talks and discussions were held with representatives of RCA, Raytheon, NAVORD, and UNIVAC while working with the AEGIS TECHREP office at Moorestown, New Jersey. The discussions established three areas that must be approached in order to insure the high reliability and availability of computer programs required in the Fleet. The first, during the design and development stage, has been previously presented. The next two areas are fault prevention and fault detection/correction after the operational programs have been placed aboard ship.

Having followed the techniques of good design, documentation, and debugging, the program master tapes that are sent to the Fleet must be assumed to be without errors. Some standard methods to insure

that mechanical faults do not enter the system should be followed. The master tapes should not be loaded to maximum density and a slow tape speed should be used. This will reduce the possibility of load errors and access errors from the tape. In addition the master tapes should be periodically checked and replaced if they show any signs of wear. While the tapes are being loaded into core and onto disc, there are several checks that should be made to see if any faults are occurring during the loading process. Parity checks on each word as it is loaded and checksums on each module will give assurance that each word and the entire module had been loaded correctly. The mechanical checks and safety precautions coupled with the program checksum and parity checks would assure that the program has been successful in moving in correct form from cards to tape into the computer. Prior to the loading of the program into the computer, the computer diagnostic programs should be run on all computers in the system to insure that they are in correct mechanical condition. Starting with the hardware in an "up" status, the correct programs, and using proper loading techniques, will reduce to a minimum the number of errors that the detection system will have to find that are not caused by the operating system. This leads to the last area of concern, and the most important: The detection of and correction of system errors that are caused by transients occurring while the system is operating.

Fault Detection/Correction

Fault detection in computer programs must satisfy two major requirements in order to be successful. It must work with a coverage approaching 100%, and it must not be excessive in the use of main processor time and core memory. The two categories that the checks fall into are core checks and operability checks. In addition, the two main causes of errors in computer programs are: 1) a transient occurs during the read/write cycle of the computer, and an improper word is written back into core; 2) a transient occurs while data is being passed around the computer from one module to another and an improbable action results. To counter the first of these problems there are several procedures that the executive writer should allow for. Whenever a module is loaded into the computer, both parity and checksums should be checked on the module. In addition, on a continuing and random basis, the checksum analysis should be made on those modules that are resident and non-changing in core. Also on a continuing and random basis, all of the instructions in core should be checked to insure that they are legal instructions for the computer. To catch these errors on an operational level, some of the program acceptance tests, suitably modified, can be run to see if the correct results are obtained for the specific inputs and tests used [Ref. 5]. The online programs for hardware fault detection in the computer should also be run. This would lead to the detection of any faults that could cause transients and affect the quality of the program in core.

The second area where program errors can occur is faults in module operations caused by transients effecting data flow. These faults are detectable in several ways, two of them inherent to the computer. If there is an overflow resulting from any form of mathematical operation, a machine interrupt is generated. An interrupt is also generated if a module attempts to address a region of core beyond its legal limits. The information that these errors occurred must go to the interrupt handling routines [Ref. 5]. These routines will determine what steps must be taken to recover from the errors generated. Other types of detection that must be implemented include the use of:

1) Arithmetic accuracy--a check to be sure that the arithmetic logic of the computer is operating correctly.
2) Reasonableness checks--to insure that the values coming into the modules are reasonable and that the results of any operations performed are also reasonable.
3) Reasonable execution time--checks to insure that the module completes its assigned task in the amount of time allotted to it.

Once an error of any type is detected, the problem of what is to be done can be solved in three ways.

An error that occurs because of improper data transfer and is detected by an interrupt will be overcome by re-initiating the request for the data needed and reentering the affected module. If the error

re-occurs, or if it is an addressing or instruction error as determined by any of the above checks, then the module that is bad should be re-loaded by itself if this is possible (dynamic reloading), re-initialized, and the operation continued as before (program rollback). Should the module be one of a complex group such that it cannot be dynamically reloaded, then the fact that this module is down must be made known to the operator, and the system must be automatically re-configured so as not to use this module.

The problem is how to implement these detection and repair processes, and several possible techniques that may be used are presented in the following paragraphs.

Parity checks should be used to check each word as it is passed into core memory from disc. As long as analysis indicates that most errors are single bit errors, then parity checks will detect all faults. The problem with the use of parity is the high amount of redundancy required in order to gain a high level of confidence in the results. James Martin [Ref. 1] discusses different aspects and applications of parity checks, with the conclusion that it must be used in conjunction with other detection methods to obtain the coverage that is desired. Also discussed are polynomial codes that can be used for the verification of input data to the system or from module to module. This use for polynomial codes is presently being used by several manufacturers for the verification of disc storage. Considering the types of errors anticipated, this would prove to be a very effective procedure to

implement. However, the present complexity of conducting software polynomial checks would lead to extensive use of machine time, or require the building of special purpose hardware to do the checking. The tradeoff of polynomial checks by special hardware, and parity checks plus additional detection methods must be made on a system by system basis.

The use of checksums to detect errors requires that the modules be produced with this in mind starting with the first design step. The sections of the module that are non-changing must be separated from the data sections so that checksums can be applied effectively. This will require additional coding, but is a very effective method of determining if a module had been changed either while being read into core or while standing in core waiting to be used. Checksum methods can be applied to operation codes and to the address portions of instructions and data accesses, being limited only by the amount of main processor time that can be dedicated to these checks.

Data reasonableness and data consistency checks must be made on all inputs to the individual modules. As part of the executive program, all data that is received into the system and all data that is passed from one module to another should be checked against predetermined limits. If the limits are exceeded, then the data should be discarded, and attempts made to acquire new data [Ref. 5]. If this attempt fails, then the system must be reconfigured to avoid

using the faulty data and the module that is presenting it. These checks would also reject information coming in a sequence that is not in line with what is expected by the executive program.

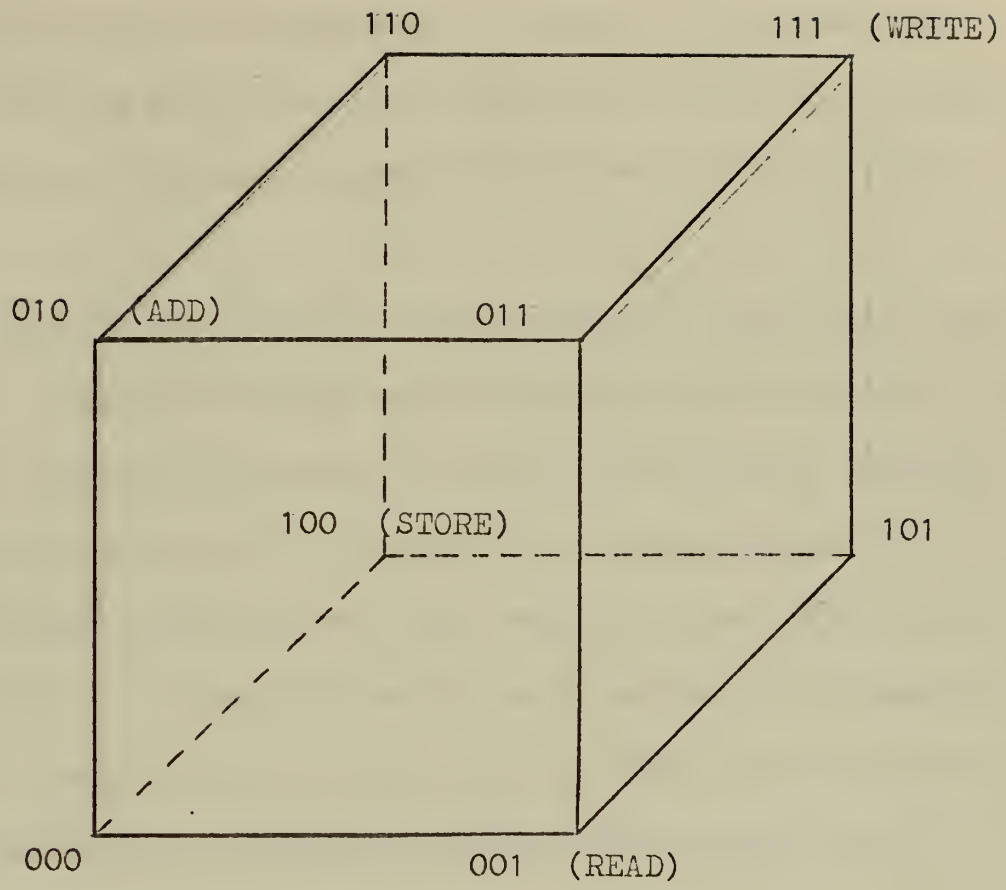
Periodic checks, though less frequent than either parity or check-sums, must be run to compare the copies of the modules in core with the copies that exist in reference storage [Ref. 9]. Again this is to verify that the unused version has not been changed while standing-by in core. Checks must also be made of the reference and it must be replaced as required by wear.

Detection of timeouts, message code errors, clock errors, and hardware parameter errors that do not result in detectable hardware errors (by generating interrupts), must be detected by the executive program (by observing performance). These are the types of transient faults that can change the quality of the program in core without affecting the hardware sensors. Attention from the start of the programming project is required to be sure that this additional coding is included as part of the computer programs.

The methods that are outlined above are applicable to any computer and machine language that is currently in use today. They must be anticipated and required during the initial design stages as part of the specifications of the computer system. The next big leap ahead in the detection of errors will be in the modification of machine languages to facilitate the detection and correction of faults. Although not as yet applicable, a possible method of implementation is described below.

The purpose of this method of machine coding would be to give as much separation as possible to the most commonly used operation codes. In addition it would separate all op codes by a minimum of two bits changes from one another. This would allow the detection of all illegal operation codes (under the assumption that transient errors would normally affect only one bit at a time) and permit the executive program to address a correct copy of the module and correct the error. In addition, multiple bit errors would be indicated by the quantity and types of instructions occurring. If an instruction is seen to be occurring at unreasonable rates, the executive program could again check the module against the reference copy and locate and correct any errors.

As a simple example consider a machine that has only four instructions, and uses three bits to implement these instructions in machine code. The instructions used are read, write, store, and add. Assuming the read and write are the most common to occur, they would get the most separation, with minimum separation being two bit changes between any instruction. Figure 2 shows the separation possible using a cube to represent the eight possible codes using three bits. The implementation of this type of checking would require the design to become effective during initial design of the computer to be used.



MACHINE CODE WITH MAXIMUM SEPARATION
FOR FAULT DETECTION

FIGURE 2

Although this method of operation code error detection is highly redundant in its coding, for systems that require extreme accuracy and high percentage fault detection, it provides 100% detection of single faults and gives an excellent indication of double faults through the use of expected execution frequency for the different operation codes.

The specific code chosen avoids using 000 as a legal code. When core for a program is allotted and data and instructions inserted, 000 is a very common core constant or leader on a data word. Avoiding its use as an opcode provides for more protection and safety from confusion when making operation code validity checks. There is much more work yet to be done before this can be applied to any operational system. The basic requirement for the rewriting of the machine language may prevent this procedure from being used until a new computer and operating system is designed.

The techniques described (parity, checksum, operation code spreading) all have one detail in common. They all make use of redundancy in coding to accomplish their checks. As redundancy has had to be applied to hardware to make it reliable, so must redundancy be applied to the computer programs in a system. The balance that is being searched for is one that provides the required availability without using excessive (expensive, time consuming) redundancy. As systems become larger, more work must be done to insure that the

system is made effective and reliable by the proper balance of redundancy techniques against performance requirements.

As an up to date example of what is being done along these lines, the SAFEGUARD system is again discussed.

The system provides effective error detection and response features within the operating system and the tactical programs. The purpose of this code is to detect errors, isolate the cause, confine the error effect, continue the program operation, and notify the system operators of the faults. Some of the error control features are:

1) A Maintenance and Diagnostic dystem provides for the automatic detection of hardware faults using an off-line support computer.
2) Detection of timeouts, data parity errors, message code errors, illegal operation codes, clock errors, and limit sensing errors are built into the hardware and are dealt with by software or by the operator as appropriate.
3) A program known as the Process Coordinator attempts to isolate the detected fault to a replaceable rack or software function and maintains maximum capability by restart or reconfiguration.
4) The code in core is compared to a reference to insure its accuracy.

5) Defensive programming techniques are used throughout--

--special code is incorporated into the modules to maintain sanity in case they are interrupted for exceeding their time limit

-- when errors are encountered, unreliable data flags are set to keep succeeding modules from using bad data

--data reasonableness checks are used extensively to test if data are within pre-established bounds.

6) Audits on critical data sets are embedded in code as required, Audit Programs, written to check global data sets for consistency, can detect certain types of error occurrences which may otherwise remain undetected until the potential impact is critical.

The SAFEGUARD System thus employs many of the new techniques that are just being developed in order to obtain the reliability needed for tactical computer systems.

VI. CONCLUSIONS

Many techniques have been described that can be used to make computer systems reliable and available. The prime consideration that has to be made is what level of performance is required of the system based on the tactical usage. If it is to be used where the correct result is of prime need, then speed can be sacrificed for the installation of all of the fault detection and correction devices described. On the other hand, if some type of operation, even in a degraded mode, is needed, then some of these devices can be left out in exchange for a very large, automatic system reconfigurer. No single statement of purpose can be made about which method is the best. The tactical requirements must be evaluated, and the amount of time, money and machine space available for fault detection and correction devices examined to determine which methods are to be used.

More time must be spent gathering data on the present Navy systems to see in which area the errors are occurring, what steps must be taken to correct the errors, and which methods are the most feasible to apply to current systems. No effective information is being gathered at present, so that any conclusions drawn about the current systems would not reflect accurately upon the situation. NELC is currently conducting research in this area [Ref. 19], and will soon have information concerning the ability to predict the

availability of tactical programs. To the knowledge of the writer, there are no programs underway investigating the need for on-line fault detection in the system computer programs. As stated before, the programs and the hardware are becoming more and more reliable, and the big problem of transient faults causing errors must be investigated to keep tactical programs running reliably with the required availability.

Future research and development work must be conducted in all areas of computer program development. The many computer societies now in existence are beginning to devote more attention to fault-tolerant software and systems. The Navy as a whole is lagging in this area. In particular, the need exists for the establishment of both general and particular specifications and standards for computer programs. Just as reliability and maintainability requirements are placed on the hardware of a system, so must they be outlined for the system software. The developers would then know what is expected of them, and different procurement efforts would conform with each other.

Programs must be instituted to gather data on the exact causes of failures to enable programmers to determine whether the failures occur because of improperly written programs or because of intermittent hardware failures, or as contended in this thesis, by transient and external abberations (hardware and software interaction).

From the management point of view several new techniques are under development and are coming into use. They include the "thread" concept [Ref. 20] of program organization as used by CSC on the AEGIS project and the creation of special program control groups as Bell Labs is doing with the SAFEGUARD program. Both of these systems appear promising and should be evaluated as the two projects continue. Rather than attempting to do development in many separate communities, the Navy should centralize the development effort to save time, money and the duplication of effort.

Of even more interest is the development of the automated program checkers and provers. Much work is being done in the civilian community that warrants detailed investigation. The algorithms for Navy tactical programs are of a type that is appropriate to this method of proof-large quantities and many repetitions of basic mathematical processes. Additional work is being done in the area of hardware design. Larger and faster discs are being developed that, when coupled with third generation computers and properly designed operating systems, will enable the use of extensive check programs without loading down the central processor with data transfers. The "new" developments in operating systems (multi-processing, shared memories) must be used extensively as they permit massive recovery and reconfiguration in tactical computer systems. These new developments will soon put present and planned operating systems out of date

unless time is now devoted to planning for the next generation of tactical systems.

The most important step that has to be taken with present tactical systems is in the information gathering area. Before any of the new practices and developments can be applied to these systems, an actual measure of their current performance and problems must be gathered. Funds must be made available so that failure information from operational systems can be gathered and analyzed. There are monitors and measuring systems currently available that would provide some of this needed information. Software and hardware devices for the Navy's specific systems must be developed and distributed to the operating ships. Once it is determined where the errors are coming from, then the techniques outlined in the previous chapters can be used as required to "failureproof" new operational systems by making them selfcontained in the fault detection and correction field.

This thesis has presented and developed some of the newer problem areas in tactical computer systems that are facing the fleet today. The developments that are taking place, the new practices that are state-of-the-art, and some possible new approaches have been outlined. There is a large gap between what exists and what can be done and what should be done. It is in this gap that future research, using some of the ideas presented in this paper as background, must be conducted.

REFERENCES

1. Martin, J., Teleprocessing Network Organization, p. 79-90, Prentice-Hall, 1970.

2. Avizienis, A., et al, "The STAR (Self-Testing and Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design", IEEE Transactions on Computers, Vol. C-20, Number 11, p. 1312-1321, November, 1971.

3. Bouricius, W., et al, "Reliability Modeling Methods for Fault-Tolerant Computing", IEEE Transactions on Computers, Vol. C-20, Number 11, p. 1306-1311, November, 1971.

4. King, J. C., "Proving Programs to be Correct", IEEE Transactions on Computers, Vol. C-20, Number 11, p. 1331-1336, November 1971.

5. Bell Laboratories letter from T. H. Crowley to LT. S. L. Margolis of April 28, 1972.

6. Avizienis, A., "Design of Fault-Tolerant Computers", Proceedings of the Fall Joint Computer Conference, Fall, 1967.

7. Watson, R. W., Timesharing System Design Concepts, p. 233-244, McGraw-Hill, 1970.

8. Bennetts & Lewin, "Fault Diagnosis of Digital Systems - A Review", Computer, Vol. 4, Number 4, July/August 1971.

9. Avizienis, A., "Fault-Tolerant Computing, An Overview", Computer, Vol. 4, Number 1, January/February 1971.

10. Carter & Bouricius, "A Survey of Fault-Tolerant Architecture & Its Evaluation", Computer, Vol. 4, Number 1, January/February 1971.

11. Elspas, Green & Levitt, "Software Reliability", Computer, Vol. 4, Number 1, January/February 1971.

12. Simmons, D., "The Art of Writing Large Programs", Computer, Vol. 5, Number 2, January/February 1972.

13. London, R., "Software Reliability Through Proving Programs Correct", Digest, 1971 International Symposium on Fault-Tolerant Computing.

14. Elspas, et al, "A Comparison of Formal Program Validation Techniques", Digest, 1971 International Symposium on Fault-Tolerant Computing.

15. Manna & Waldinger, "Toward Automatic Program Synthesis", Communications of ACM.

16. Dijkstra, E., "A Constructive Approach to the Problem of Program Correctness", BIT, Vol. 8, Number 3, 1968.

17. Naur, P., "Programming by Action Clusters", BIT, Vol. 9, Number 3, 1969.

18. von Neuman, J., & Goldstine, H., Planning and Coding Problems for an Electronic Computing Instrument", "Collected Works of John von Neumann", Pergamon Press, 1961.

19. Schneidewind, N., "A Methodology for Software Reliability Prediction and Quality Control", NPS55SS72032B, March, 1972.

20. "Thread Oriented Configuration Control System", Preliminary Draft, Computer Science Corporation, Moorestown, New Jersey, Summer, 1971.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3.	Asst Professor V. M. Powers, Code 52Pw Department of Electrical Engineering Naval Postgraduate School Monterey, California 93940	2
4.	Lt. Sheldon Lee Margolis, USN 5606 Bland Avenue Baltimore, Maryland 21215	1
5.	Naval Ordnance Systems Command Technical Representative (AEGIS) RCA Moorestown, New Jersey 08057 Attn: Commander W. Phillips	1

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)

Naval Postgraduate School
Monterey, California 93940

2a. REPORT SECURITY CLASSIFICATION

Unclassified

2b. GROUP

3. REPORT TITLE

Computer Program Fault Detection and Correction

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Master's Thesis; June 1972

5. AUTHOR(S) (First name, middle initial, last name)

Sheldon Lee Margolis

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
June 1972	56	20

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
b. PROJECT NO.	
c.	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
d.	

10. DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Naval Postgraduate School Monterey, California 93940

13. ABSTRACT

A major problem with present fleet tactical computer programs is the failures that occur because of software faults caused by intermittent transient errors. Presented are measures that should be taken to prevent the generation of these transients, and possible steps for the detection and the correction of faults caused by these transients. Examples are drawn from the practices used in the SAFEGUARD, AEGIS, and STAR computer system programs. The paper describes some of the causes of program degradation and requirements for future activities to overcome these problems. Some specific examples are shown to illustrate corrective measures that could be taken. The applicability of protective redundancy to computer programs is established.

DD FORM 1 NOV 65 1473 (PAGE 1)
S/N 0101-807-6811

A-31408

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Computer program fault detection						
Computer program fault prevention						
SAFEGUARD Fault detection and prevention						
Hardware redundancy						

135184
Thesis
M3425 Margolis
c.1 Computer program fault
detection and correction.
S1060
26750
33073
29 AUG 84

Thesis
M3425 Margolis
c.1 Computer program fault
detection and correction.

135184

thesM3425
Computer program fault detection and cor
3 2768 001 03411 9
DUDLEY KNOX LIBRARY